CS 267 Applications of Parallel Computers

Lecture 6:
Distributed Memory (continued)

Data Parallel Architectures and Programming

Bob Lucas

Based on previous notes by James Demmel and David Culler

www.nersc.gov/~dhbailey/cs267

Recap of Last Lecture

° **Distributed memory machines**

- Each processor has independent memory
- Connected by network
  - topology, other properties

° **Cost =**

#messages * α + #words_sent * β + #flops * f + delay

° **Distributed memory programming**

- MPI
- Send/Receive
- Collective Communication
- Sharks and Fish under gravity as example

Outline

- **Distributed Memory Programming (continued)**
  - **Review Gravity Algorithms**
  - **Look at Sharks and Fish code**
- **Data Parallel Programming**
  - **Evolution of Machines**
  - **Fortran 90 and Matlab**
  - **HPF (High Performance Fortran)**

Example: Sharks and Fish

° **N fish on P procs, N/P fish per processor**

- **At each time step, compute forces on fish and move them**

° **Need to compute gravitational interaction**

- **In usual N^2 algorithm, every fish depends on every other fish**

$$\text{force on } j = \sum_{\substack{k=1:N \\ k \,!= j}} (\text{force on } j \text{ due to } k)$$

- **every fish needs to "visit" every processor, even if it "lives" on one**

° **What is the cost?**

2 Algorithms for Gravity: What are their costs?

Algorithm 1

```
Copy local Fish array of length N/P to Tmp array
for j = 1 to N
    for k = 1 to N/P,   Compute force from Tmp(k) on Fish(k)
    "Rotate" Tmp by 1
        for k=2 to N/P, Tmp(k) <= Tmp(k-1)
        recv(my_proc - 1,Tmp(1))
        send(my_proc+1,Tmp(N/P)
```

Algorithm 2

```
Copy local Fish array of length N/P to Tmp array
for j = 1 to P
    for k=1 to N/P, for m=1 to N/P, Compute force from Tmp(k) on Fish(m)
    "Rotate" Tmp by N/P
        recv(my_proc - 1,Tmp(1:N/P))
        send(my_proc+1,Tmp(1:N/P))
```

What could go wrong? (be careful of overwriting Tmp)

More Algorithms for Gravity

- **Algorithm 3 (in sharks and fish code)**
  - All processors send their Fish to Proc 0
  - Proc 0 broadcasts all Fish to all processors
- **Tree-algorithms**
  - Barnes-Hut, Greengard-Rokhlin, Anderson
  - O(N log N) instead of O(N^2)
  - Parallelizable with cleverness
  - "Just" an approximation, but as accurate as you like (often only a few digits are needed, so why pay for more)
  - Same idea works for other problems where effects of distant objects becomes "smooth" or "compressible"
    - electrostatics, vorticity, …
    - radiosity in graphics
    - anything satisfying Poisson equation or something like it
  - May talk about it in detail later in course

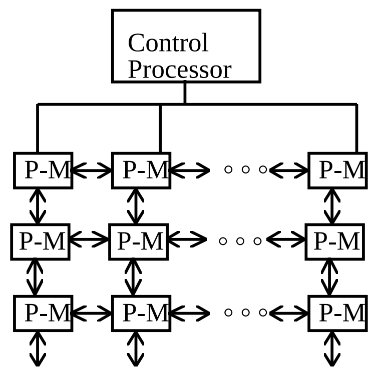
Examine Sharks and Fish Code

- www.cs.berkeley.edu/~demmel/cs267_Spr99/Lectures/fish.c

Data Parallel Machines

Data Parallel Architectures

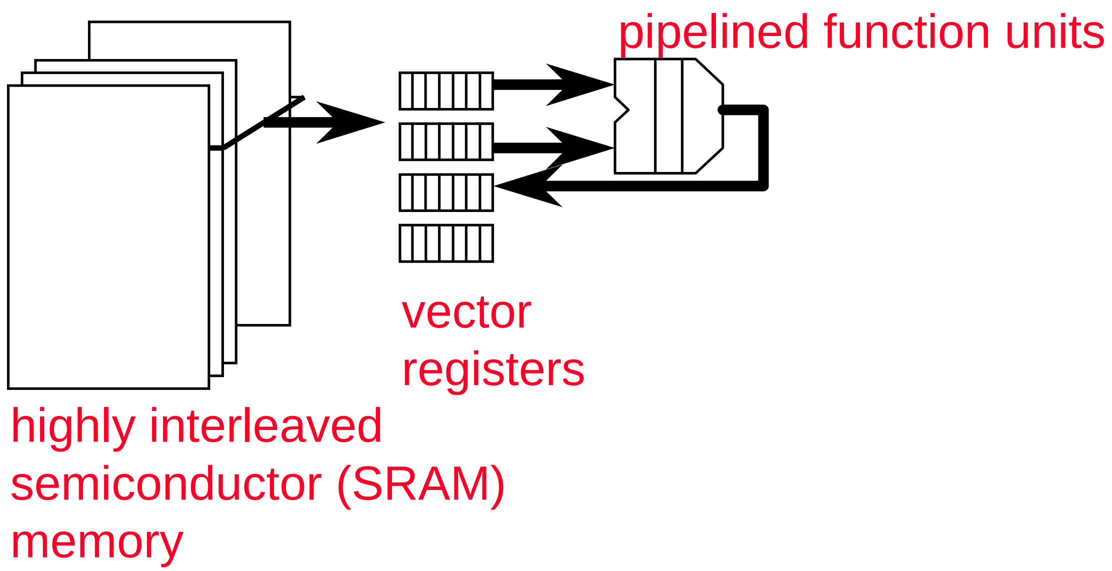
- Programming model
  - operations are performed on each element of a large (regular) data structure in a single step
  - arithmetic, global data transfer
- A processor is logically associated with each data element
  - A=B+C means for all j, A(j) = B(j) + C(j) in parallel
- General communication
  - A(j) = B(k) may communicate
- Global synchronization
  - implicit barrier between statements
- SIMD: Single Instruction, Multiple Data



Vector Machines

° **The Cray-1 and its successors (www.sgi.com/t90)**

- **Load/store into 64-word Vector Registers, with strides: vr(j) = Mem(base + j*s)**
- **Instructions operate on entire vector registers: for j=1:N vr1(j) = vr2(j) + vr3(j)**



° **No cache, but very fast (expensive) memory**

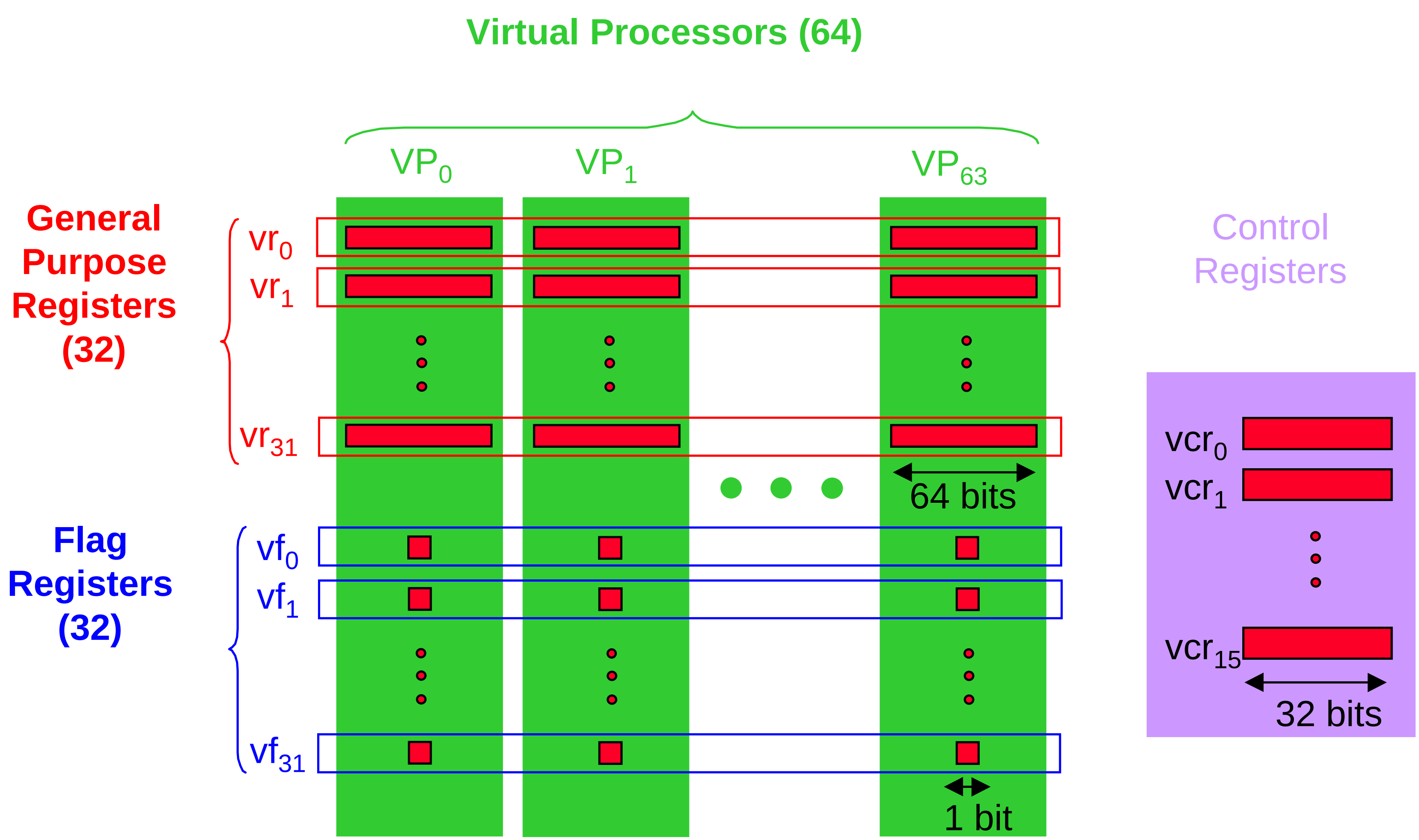
° **Scatter [Mem(Pnt(j)) = vr(j)] and Gather [vr(j) = Mem(Pnt(j)]**

° **Flag Registers [vf(j) = (vr3(j) != 0)]**

° **Masked operations [vr1(j) = vr2(j)/vr3(j) where vf(j)==1]**

° **Fast scalar unit too**

Use of SIMD Model on Vector Machines



Evolution of Vector Processing

- Cray (now SGI), Convex, NEC, Fujitsu, Hitachi,…
- Pro: Very fast memory makes it easy to program
  - Don't worry about cost of loads/stores, where data is (but memory banks)
- Pro: Compilers automatically convert loops to use vector instructions
  - for j=1 to n, A(j) = x*B(j)+C(k,j) becomes sequence of vector instructions that breaks operation into groups of 64
- Pro: Easy to compile languages like Fortran90
- Con: Much more expensive than bunch of micros on network
- Relatively few customers, but powerful ones
- New application: multimedia
  - New microprocessors have fixed point vector instructions (MMX, VIS)
  - VIS (Sun's Visual Instruction Set) (www.sun.com/sparc/vis)
    - 8, 16 and 32 bit integer ops
    - Short vectors only (2 or 4)
    - Good for operating on arrays of pixels, video

Data parallel programming

Evolution of Data Parallel Programming

° Early machines had single control unit for multiple arithmetic units, so data parallel programming was necessary

° Also a natural fit to vector machines

° Can be compiled to run on any parallel machine, on top of shared memory or MPI

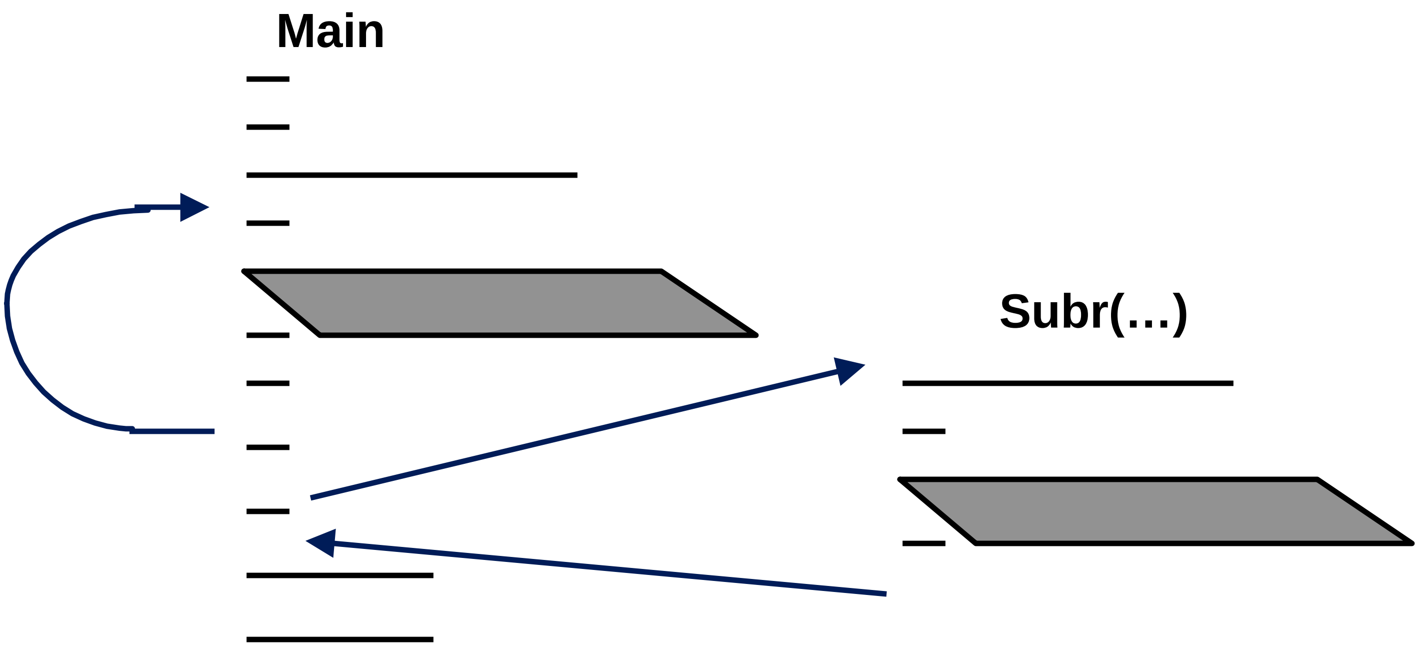
° Fortran 77

  -> Fortran 90

    -> HPF (High Performance Fortran)

Fortran90 Execution Model (also Matlab)

- Sequential composition of parallel (or scalar) statements
- Parallel operations on **arrays**



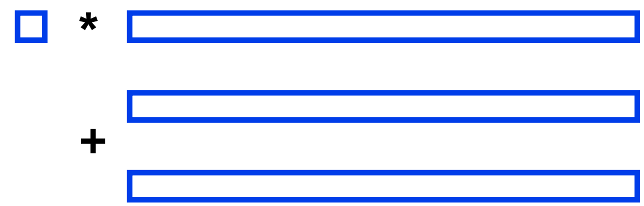
- Arrays have **rank** (# dimensions), **shape** (extents), **type** (elements)
  - HPF adds **layout**
- Communication implicit in array operations
- Hardware configuration independent

Example: gravitational fish

parallel assignment

pointwise parallel operator

```
    integer, parameter :: nfish = 10000
    complex fishp(nfish), fishv(nfish), force(nfish), accel(nfish)
    real    fishm(nfish)
. . .
    do while (t < tfinal)
       t = t + dt
       fishp   = fishp + dt*fishv
       call compute_current(force,fishp)
       accel   = force/fishm
       fishv   = fishv + dt*accel
       ...
    enddo
. . .
    subroutine compute_current(force,fishp)
    complex force(:),fishp(:)
    force = (3,0)*(fishp*(0,1))/(max(abs(fishp),0.01)) - fishp
    end
```



Array Operations

Parallel Assignment

A = 0	! scalar extension
L = .TRUE.	
B = [1,2,3,4]	! array constructor
X = [1:n]	! real sequence [1.0, 2.0, . . .,n]
I = [0:100:4]	! integer sequence [0,4,8,...,100]
C = [50[1], 50[2,3]]	! 150 elements, first 1s then repeated 2,3
D = C	! array copy

Binary array operators operate pointwise on **conformable** arrays

- have the same size and shape

Array Sections

Portion of an array defined by a triplet in each dimension

- **may appear wherever an array is used**

```
A(3)            ! third element
A(1:5)          ! first five elements
A(1:5:1)        ! same
A(:5)           ! same
A(1:10:2)       ! odd elements in order
A(10:2:-2)      ! even in reverse order
A(10:2:2)       ! same

B(1:2,3:4)      ! 2x2 block
B(1, :)         ! first row
B(:, j)         ! jth column
```

Reduction Operators

Reduce an array to a scalar under an associative binary operation

- sum, product
- minval, maxval
- count (number of .TRUE. elements of logical array)
- any, all

simplest form of communication

implicit broadcast

```
do while (t < tfinal)
   t = t + dt
   fishp    = fishp + dt*fishv
   call compute_current(force,fishp)
   accel    = force/fishm
   fishv    = fishv + dt*accel
   fishspeed = abs(fishv)
   mnsqvel = sqrt(sum(fishspeed*fishspeed)/nfish)
   dt       = .1*maxval(fishspeed) / maxval(abs(accel))
enddo
```

Conditional Operation

```
force = (3,0)*(fishp*(0,1))/(max(abs(fishp),0.01)) - fishp
```

could use

```
dist = 0.01
where (abs(fishp) > dist) dist = abs(fishp)
```

or

```
far = abs(fishp) > 0.01
where far dist = abs(fishp)
```

or

```
where (abs(fishp) .ge. 0.01)
    dist = abs(fishp)
elsewhere
    dist = 0.01
end where
```

No nested wheres. Only assignment in body of the where.
The boolean expression is really a mask array.

Forall in HPF (Extends F90)

FORALL (triplet, triplet,…,mask) assignment

```
forall ( i = 1:n) A(i) = 0   ! same as A = 0
forall ( i = 1:n ) X(i) = i  ! same as X = [ 1:n ]
forall (i=1:nfish) fishp(i) = (i*2.0/nfish)-1.0

forall (i=1:n, j = 1:m) H(i,j) = i+j
forall (i=1:n, j = 1:m) C(i+j*2) = j

forall (i = 1:n) D(Index(i)) = C(i,i)    ! Maybe
forall (i=1:n, j = 1:n, k = 1:n)
*     C(i,j) = C(i,j) + A(i,k) * B(k,j)   ! NO
```

Evaluate entire RHS for all index values (in any order)

Perform all assignments (in any order)

No more than one value for each element on the left (may be checked)

Conditional (masked) intrinsics

Most intrinsics take an optional mask argument

```
funny_prod = product( A, A .ne. 0)
bigem = maxval(A, mask = inside )
```

Use of masks in the FORALL assignment (HPF)

```
forall ( i=1:n, j=1:m, A(i,j) .ne. 0.0 ) B(i,j) = 1.0 / A(i,j)
forall ( i=1:n, inside) A(i) = i/n
```

Subroutines

- Arrays can be passed as arguments.
- Shapes must match.
- Limited dynamic allocation
- Arrays passed by reference, sections by value (i.e., a copy is made)
    - HPF: either remap or inherit
- Can extract array information using inquiry functions

Implicit Communication

Operations on conformable array sections may require data movement

i.e., communication

```
A(1:10, : ) = B(1:10, : ) +  B(11:20, : )
```

Example: Parallel finite differences

```
A’[i] = (A[i+1] - A[i])*dt   becomes
A(1:n-1) = (A(2:n) - A(1:n-1)) * dt
```

Example: smear pixels

```
show(:,1:m-1) = show(:,1:m-1) + show(:,2:m)
show(1:m-1,:) = show(1:m-1,:) + show(2:m,:)
```

Global Communication

```
c(:, 1:5:2) = c(:, 2:6:2)     ! shift noncontiguous sections

D = D(10:1:-1)                ! permutation (reverse)

A = [1,0,2,0,0,0,4]
I = [1,3,7]
B = A(Ind)                    ! Ind = [1,2,4]   "gather"
C(Ind) = B                    ! C = A  "scatter" (no duplicates on left)

D = A([1,1,3,3])              ! replication
```

Specialized Communication

CSHIFT(array, dim, shift) **! cyclic shift in one dimension**

EOSHIFT(array, dim, shift [, boundary]) **! end off shift**

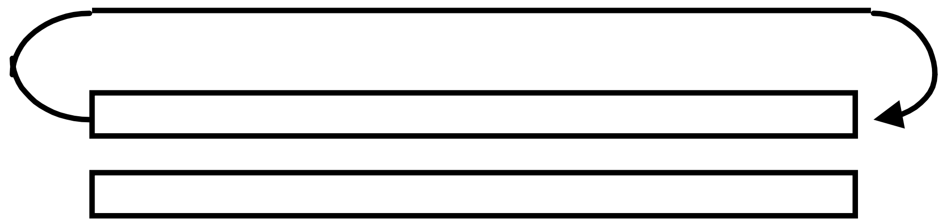
TRANSPOSE(matrix) **! matrix transpose**

SPREAD(array, dim, ncopies)

Example: nbody calculation

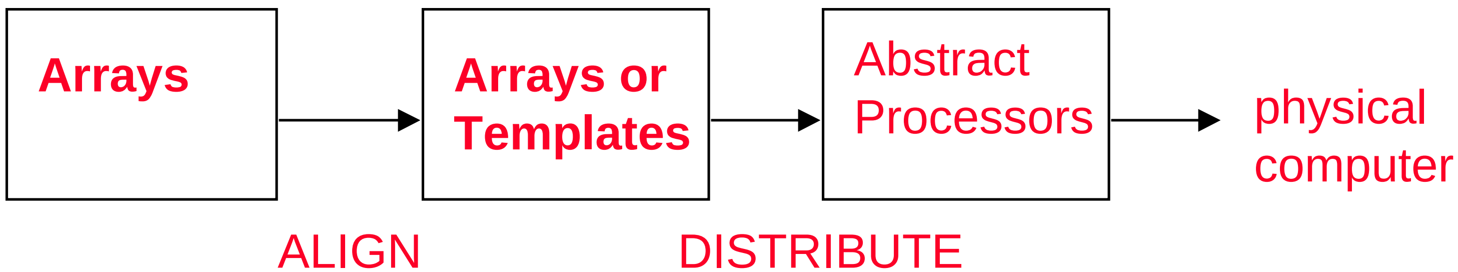
```
subroutine compute_gravity(force,fishp,fishm,nfish)
  complex force(:),fishp(:),fishm(:)
  complex fishmp(nfish),  fishpp(DSHAPE(fishp)), dif(DSIZE(force))
  integer k

  force  = (0.,0.)
  fishpp = fishp
  fishmp = fishm
  do k=1, nfish-1
    fishpp = cshift(fishpp, DIM=1, SHIFT=-1)
    fishmp = cshift(fishmp, DIM=1, SHIFT=-1)
    dif = fishpp - fishp
    force = force + (fishmp * fishm * dif / (abs(dif)*abs(dif)))
  enddo
  end
```

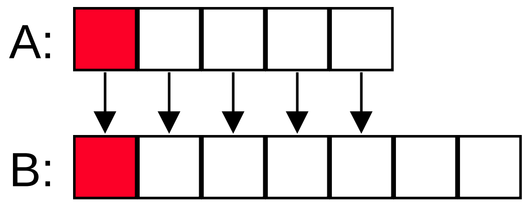


HPF Data Distribution (layout) directives

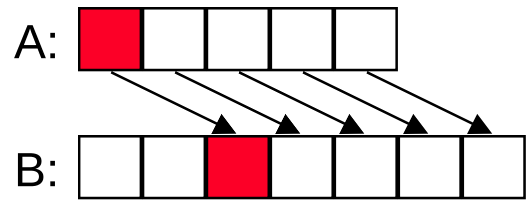
- **Can ALIGN arrays with other arrays for affinity**
  - **elements that are operated on together should be stored together**
- **Can ALIGN with TEMPLATE for abstract index space**
- **Can DISTRIBUTE templates over processor grids**
- **Compiler maps processor grids to physical procs.**



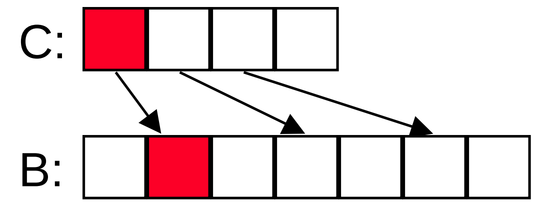
Alignment



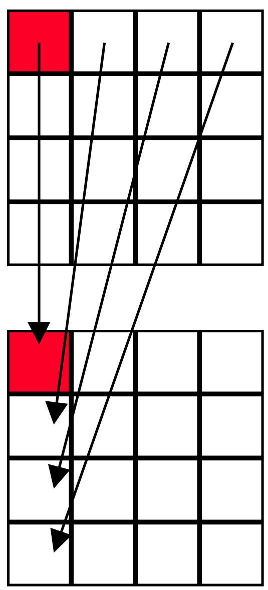
ALIGN A(I) WITH B(I)



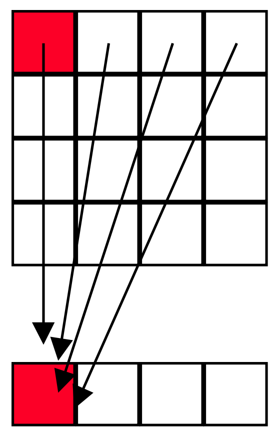
ALIGN A(I) WITH B(I+2)



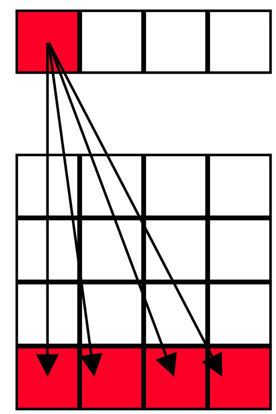
ALIGN C(I) WITH B(2*I)



ALIGN D(i,j) WITH E(j,i)



ALIGN D(:,*) with A(:)
- collapse dimension

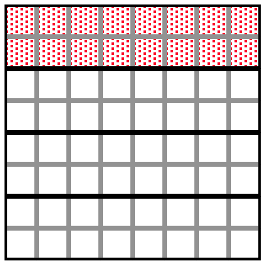


?

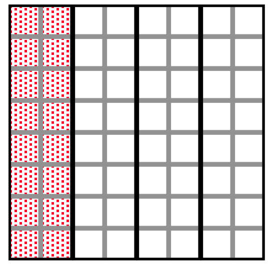
ALIGN A(:) with D(*,:)
- replication

Layouts of Templates on Processor Grids

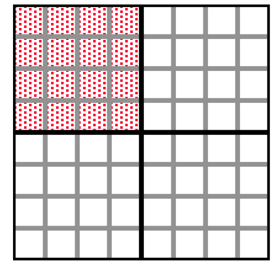
° **Laying out T(8,8) on 4 processors**



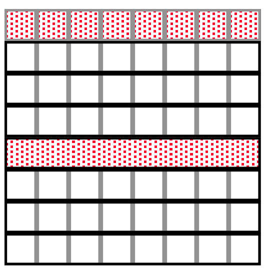
(Block, *)



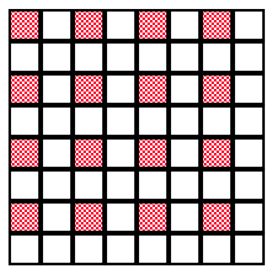
(*, Block)



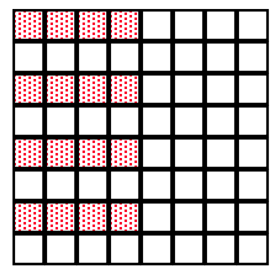
(Block, Block)



(Cyclic, *)



(Cyclic, Cyclic)



(Cyclic, Block)

Example Syntax

Declaring Processor Grids

```
!HPF$          PROCESSORS P(32)

!HPF$          PROCESSORS Q(4,8)
```

Distributing Arrays onto Processor Grids

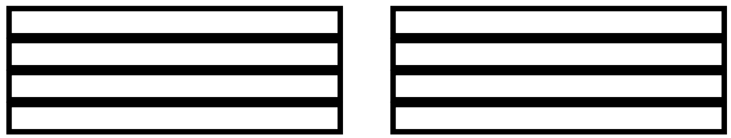
```
!HPF$          PROCESSORS p(32)

                   real D(1024), E(1024)
!HPF$          DISTRIBUTE D(BLOCK)

!HPF$          DISTRIBUTE E(BLOCK) ONTO p
```

Blocking Gravity in HPF

```
 subroutine compute_gravity(force,fishp,fishm,nblocks)
 complex force(:,B),fishp(:,B),fishm(:,B)
 complex fishmp(nblocks,B), fishpp(nblocks,B),dif(nblocks,B)
!HPF$ Distribute force(block,*), . . .
 force  = (0.,0.)
 fishpp = fishp
 fishmp = fishm
 do k=1, nblocks-1
   fishpp  = cshift(fishpp,  DIM=1, SHIFT=-1)
   fishmp = cshift(fishmp, DIM=1, SHIFT=-1)
   do j = 1, B
      forall (i = 1:nblocks) dif(i,:) = fishpp(i,j) - fishp(i,:)
      forall (i = 1:nblocks) force(i,:) = force(i,:) +
*         (fishmp(i,j) * fishm(i,:) * dif(i,:) / (abs(dif(i,:))*abs(dif(i,:))))
   end do
 enddo
```



HPF “Independent” Directive

° **Assert that the iterations of a do-loop can be performed independently without changing the result computed.**

- **Tells compiler “trust me, you can run this in parallel”**
- **In any order or concurrently**

```
!HPF$   INDEPENDENT
          do i=1,n
            A(Index(i)) = B(i)
          enddo
```

Parallel Prefix (Scan) Operations

```
forall (i=1:5) B(i) = SUM( A(1:i) )          ! forward running sum
forall (i=1:n) B(i) = SUM( A(n-i+1:n) )   ! reverse direction


dimension fact(n)
fact = [1:n]
forall (i=1:n) fact(i) = product( fact(1:i) )
```

or

CMF_SCAN_*op* (dest,source,segment,axis,direction,inclusion,mode,mask)

op = [add,max,min,copy,ior,iand,ieor]

Other Data Parallel Languages

- *LISP, C*, DPCE
- NESL, FP
- PC++
- APL, MATLAB, . . .